OS 2022 Project Testing Cases

1. Test each part from the project independently.
2. After completing all parts, test the whole project using the testing scenarios.
3. The individual tests and scenarios MUST meet the following time limits:

    1. *tstkvirtaddr (k virtual address test):* ***max of 3 min / each***
    2. *Scenarios:* ***max of 4 min / each***
    3. *All other individual tests****: max of 1 min / each***

4. During your solution, don't change any file EXCEPT those who contain "TODO",
5. In bonuses & challenges, if you change any other file during your solution, kindly MAKE SURE to tell us when you deliver the code

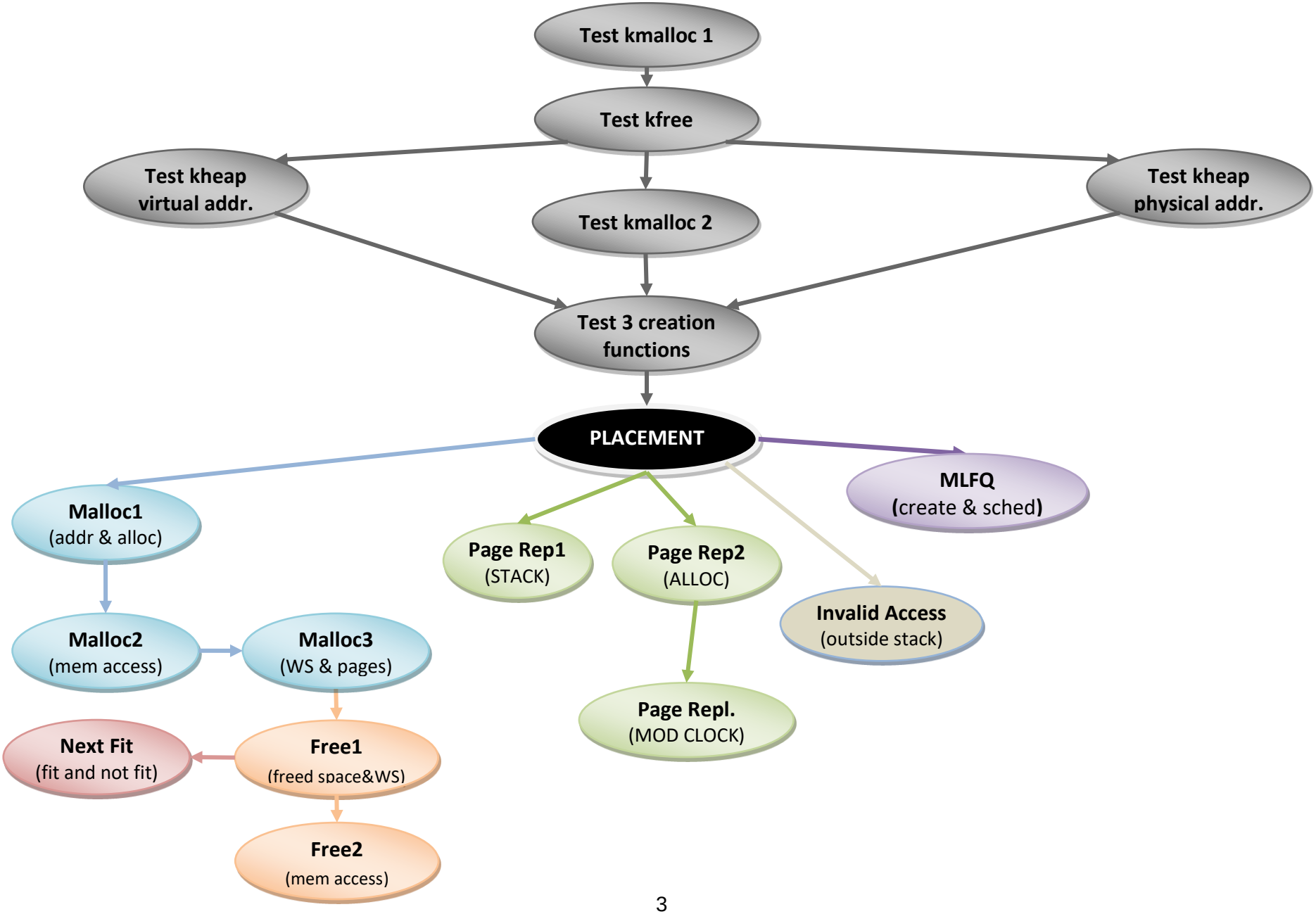
A- Dependency Graph of Ready-Made Tests

The following graph shows the dependencies between the ready-made tests.

For example:

- To test **Placement**, you first need to successfully test the following: kmalloc, kfree, kheap_virtual_address, kheap_physical_address and the 3 creation functions.
- On the other hand, testing **Kmalloc** doesn't depend on any test.

All tests are based on the page placement, which in turn is based on **KERNEL HEAP** tests. So you need to first implement the KERNEL HEAP functions and test them with the **given test program**.


Test kmalloc 1
Test kfree
Test kheap virtual addr.
Test kmalloc 2
Test kheap physical addr.
Test 3 creation functions
PLACEMENT
Malloc1 (addr & alloc)
Malloc2 (mem access)
Malloc3 (WS & pages)
Next Fit (fit and not fit)
Free1 (freed space&WS)
Free2 (mem access)
Page Rep1 (STACK)
Page Rep2 (ALLOC)
Page Repl. (MOD CLOCK)
MLFQ (create & sched)
Invalid Access (outside stack)

B- Responsibility of Each Ready-Made Test

The following tables show the main points that each of the test programs will check for!!

Kmalloc (1, 2 & 3)	Kfree	Kheap_virtual_address	Kheap_physical_address	Three creation functions
1. Return addr. (4KB boundary) 2. Memory allocation (Next Fit) 3. Page File allocation (nothing) 4. memory access (R & W) 5. Insufficient space 6. Permissions	1. Memory de-allocation 2. Tables of KHEAP (exists) 3. Memory access after free 4. Del. Non-exist variable 5. Allocation after free	1. Get va after kmalloc only 2. Get va after kmalloc & kfree 3. Get va of frames that are not belong to KHEAP	1. Get pa after kmalloc only 2. Get pa after kmalloc & kfree 3. Get pa of non-exist area	1. Create page working set array 2. Create page directory 3. Create new page table and link it to the directory

Placement	Invalid Access	Page Replace#1 (Alloc)	Page Replace#2 (Stack)	Page Replace(ModClk)	MLFQ (scheduling)
1. Updating WS & last index 2. Mem. Allocation (increased) 3. Adding new stack pages to Page File	Illegal memory access to page that's not exist in Page File and not STACK	1. Mem. Allocation (no change) 2. Page File allocation (no change)	1. Add new stack pages to Page File for 1st time ONLY, then update 2. Mem. Allocation 3. Victimize and restore stack page	1. Working set after removing ModClk pages. 2. WS last index. (No empty locations in the WS)	1. Placing the current env in its correct queue 2. Selection of the next env and setting CPU clock.

Malloc1	Malloc2	Malloc3	Free1 (with placement)	Free2 (with placement)	Next fit
1. Return addresses (4KB boundary) 2. Page File allocation 3. Memory allocation (nothing)	Memory access (read & write) of the allocated spaces	After accessing: check num of pages and WS entries	1. Deleting from page file 2. Deleting WS pages 3. Deleting empty tables 4. Updating WS	1. Clear entry of dir. & table 2. Can't access any page again (i.e. fault on it lead to invalid access)	Request allocations of variables that either fit or not fit in one of the free segments.

C- Testing Procedures

FIRST: Testing Each Part

Run every test of the following. If a test succeeds, it will print and success message on the screen, otherwise the test will panic at the error line and display it on the screen.

> **IMPROTANT NOTES:**
>
> 1. Run each test in **NEW SEPARATE RUN**
> 2. If the test of certain part failed, then there's a problem in your code
> 3. Else, this does NOT ensures 100% that this part is totally correct. So, make sure that your logic matches the specified steps exactly

1. Testing KERNEL Heap:

tstkmalloc1 command: tests the implementation of **kmalloc()**. It validates return addresses from the kmalloc(),number of allocated frames, accessing the allocated space and permissions (Continuous Allocation)

- **FOS>** `tstkmalloc 1`

tstkmalloc2 command: tests the implementation of **kmalloc()** (kfree must be implemented in order to run this test). It validates return addresses from the kmalloc(), testing the Next fit strategy by creating some holes in the memory using **kfree().**

- **FOS>** `tstkmalloc 2`

tstkfree command: tests the implementation of **kfree()**. It validates the number of freed frames by kfree(). It checks the memory access (read & write) of the removed spaces and allocation after free. Also, it ensure that KHEAP tables are not removed.

- **FOS>** `tstkfree`

tstkvirtaddr command: tests the implementation of **kheap_virtual_address()**. It validates the returned virtual address of the given physical one for three cases: 1. After kmalloc only, 2. After kmalloc and kfree, 3. For frames that does not belong to KERNEL HEAP (should return 0).

- **FOS>** `tstkvirtaddr`

tstkphysaddr command: tests the implementation of **kheap_physical_address()**. It validates the returned physical address of the given virtual one for three cases: 1. after kmalloc only, 2. after kmalloc and kfree, 3. for not allocated area in KERNEL HEAP (should return 0).

- **FOS>** `tstkphysaddr`

2. Testing Three Creations Functions:

tst3functions command: run fos_add program to test the implementations of three creations functions: create_user_page_WS(), create_user_directory() and create_page_table(). All 3 functions should do their allocations in KERNEL HEAP

- **FOS>** `tst3functions //fist time: to run fos_add`
- **FOS>** `tst3functions //second time: to test the creation in KHEAP`

---

3. Testing Page Fault Handler:

tst_placement.c (tpp): tests page faults on stack + page placement

FOS> `run tpp 20`

tst_invalid_access.c (tia): tests handling illegal memory access (request to access page that's not exist in page file and not belong to the stack) it should display the panic you have written

FOS> `run tia 15`

tst_page_replacement_alloc.c (tpr1): tests allocation in memory and page file after page replacement.

FOS> `run tpr1 11`

tst_page_replacement_stack.c (tpr2): tests page replacement of stack (creating, modifying and reading them)

FOS> `run tpr2 6`

tst_page_replacement_mod_clock.c (tmodclk): tests page replacement by MODIFIED CLOCK algorithm

FOS> `run tmodclk 11`

---

4. Testing CPU Scheduling with MLFQ:

tst_CPU_MLFQ_master_1.c (tmlfq1): tests the MLFQ method for CPU scheduling. It tests the placement of the current environment (if exist) in its correct queue. In addition, it tests the correct selection of the next process and setting the CPU clock by the correct quantum.

- `FOS> schedMLFQ 5 2 4 6 8 10`
- `FOS> run tmlfq1 100`

---

5. Testing User Heap:

tst_malloc_1.c (tm1): tests the implementation **malloc()** & **allocateMem()**. It validates both the return addresses from the malloc() and the number of allocated frames by allocateMem().

- **FOS>** `run tm1 2000`

tst_malloc_2.c (tm2): tests the implementation **malloc()** & **allocateMem()**. It checks the memory access (read & write) of the allocated spaces.

- **FOS>** `run tm2 2000`

tst_malloc_3.c (tm3): tests the implementation **malloc()** & **allocateMem()**. After accessing the memory, it checks the number of allocated frames and the WS entries.

- **FOS>** `run tm3 2000`

tst_free_1.c (tf1): tests the implementation **free()** & **freeMem()**. It validates the number of freed frames by freeMem().

- **FOS>** `run tf1 2000`

tst_free_2.c (tf2): tests the implementation **free()** & **freeMem()**. It checks the memory access (read & write) of the removed spaces.

- **FOS>** `run tf2 2000`

tst_next_fit.c (tnf): tests the **Next fit strategy** by requesting allocations that either fit of not fit in one of the free segments. Some requests should be granted while others should not.

- **FOS>** `run tnf 2000`

SECOND: Testing Whole Project

You should run each of the following scenarios successfully

Scenario 1: Running single program to Test ALL MODULES TOGETHER

REQUIRED MODULES:

1. KERNEL Heap
2. USER Heap (malloc & free)
3. Page Fault Handler (placement + replacement)

FOS> `run tqsfh 7` `//run` **`tst_quicksort_freeHeap`**

test it according to the following steps:

- Number of Elements = **1,000**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **5,000**

  Initialization method : **Descending**

  Do you want to repeat (y/n) : **y**

- Number of Elements = **300,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

  "At each step, the program should sort the array successfully"

Scenario 2: Running multiple programs with PAGES suffocation

REQUIRED MODULES:

1. KERNEL Heap
2. USER Heap (malloc only)
3. Page Fault Handler (replacement)

```
1. FOS> load fib 7          //load Fibonacci program
2. FOS> load tqs 7          //load Quick sort program [with leakage]
3. FOS> load ms2 7          //load Merge sort program [with leakage]
4. FOS> runall         //run all of them together
```

Test them according to the following steps:

[Fibonacci]

- Fibonacci index = **30** **"Result should = 1346269"**

[QuickSort]

- Number of Elements = **1,000**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**
- Number of Elements = **1,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

"At each step, the program should sort the array successfully"

[MergeSort]

- Number of Elements = **32**

  Initialization method : **Ascending**

  Do you want to repeat (y/n) : **y**
- Number of Elements = **32**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

"At each step, the program should sort the array successfully"

Scenario 3: MLFQ

REQUIRED MODULES:

1. KERNEL Heap
2. USER Heap (malloc and free)
3. Page Fault Handler (replacement and replacement)
4. MLFQ

1. **FOS>** `schedMLFQ 4 20 30 40 50`
2. **FOS>** `load qs 100`
3. **FOS>** `load fib 100`
4. **FOS>** `load tmlfq 100`
5. **FOS>** `load tmlfq 1000`
6. **FOS>** `runall` `//run both of them together`

[Fibonacci]

- Fibonacci index = **30** **"Result should = 1346269"**

[QuickSort]

- Number of Elements = **100,000**

  Initialization method : **Semi random**

  Do you want to repeat (y/n) : **n**

"The program should sort the array successfully"

THIRD: Testing Bonuses

You should run each of the following tests according to the bonus you have implemented.

1. The best fit strategy for the Kernel Heap allocations

1.1 testkmalloc1 command: tests the implementation of kmalloc(). It validates return addresses from the kmalloc(),number of allocated frames, accessing the allocated space and permissions (Continuous Allocation)

- **FOS>** khbestfit //the allocation strategy is now best fit
- **FOS>** tstkmalloc 1

A success message should be displayed

1.2 testkmalloc2 command: tests the implementation of kmalloc() (kfree must be implemented in order to run this test). It validates return addresses from the kmalloc(), testing the best fit strategy by creating some holes in the memory using kfree().

- **FOS>** khbestfit //the allocation strategy is now best fit
- **FOS>** tstkmalloc 2

A success message should be displayed

1.3 testkmalloc3 command: : tests the implementation of kmalloc(). Tests the best fit strategy by requesting allocations that can't fit in any of the free segments. All requests should NOT be granted.

- **FOS>** khbestfit //the allocation strategy is now best fit
- **FOS>** tstkmalloc 3

A success message should be displayed

2. The env_free function to free all the memory allocated for an environment

2.1 test env free without using dynamic allocation/de-allocation

- **FOS>** run tef1 10

a success message should be displayed

2.2 test env free without using dynamic allocation/de-allocation

- **FOS>** run tef2 20

a success message should be displayed

3. The user realloc function

3.1 test realloc 1

tests the reallocation that both fits and does not fit into the same location

- **FOS>** run tr1 3000

a success message should be displayed

3.2 test realloc 2

tests the special cases of reallocation:

1. Re-allocate with size = 0
2. Re-allocate with address = NULL
3. Re-allocate in the existing internal fragment (no additional pages are required)
4. Re-allocate that can NOT fit in any free fragment
5. Re-allocate that test Next FIT strategy

- **FOS>** run tr2 3000

a success message should be displayed

3.3 test realloc 3

tests the data after reallocation

- **FOS>** run tr3 3000

a success message should be displayed

4. The user program priority

3.4 test priority 1

- **FOS>** tstpriority1

Should run three programs

- **FOS>** tstpriority1

a success message should be displayed

3.5 test priority 2

- **FOS>** tstpriority2

Should run three programs

- **FOS>** tstpriority2

a success message should be displayed

FOURTH: Testing Challenges

Run the following, EACH in a SEPARATE RUN:

1. **Stack deallocations**
   Test with palcement

   - FOS> run tfs 1000

   Test with REPLACEMENT

   - FOS> run tfs 10

   In each test, it should print the successful message

Enjoy writing your own OS

☺ GOOD LUCK ☺